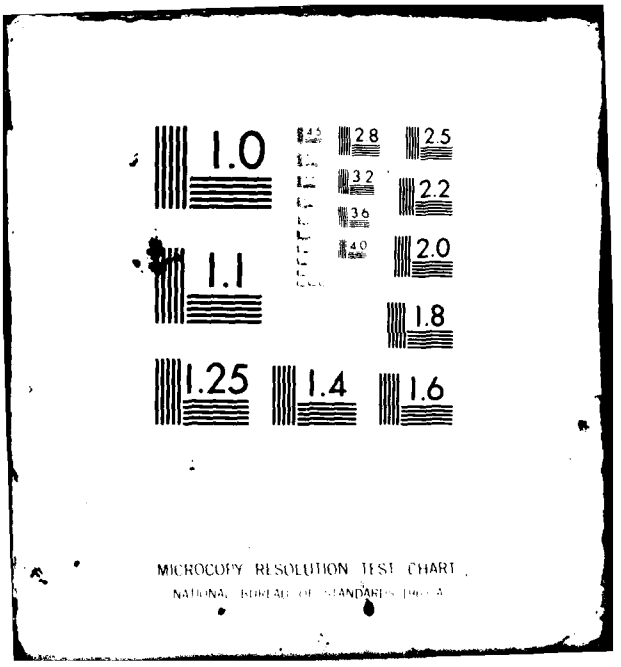
AD-A110 224 SYSTEMS CONTROL INC PALO ALTO CA COMPUTER SCIENCE DEPT F/G 9/2
CODIFICATION OF PROGRAM SYNTHESIS KNOWLEDGE FOR CONCURRENT PROG--ETC(U)
SEP 81 D CHAPIRO F49620-79-C-0137
UNCLASSIFIED SCI-ICS-L-81-1 AFOSR-TR-81-0895 NL

END
DATE FILMED
2 82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART

LEVEL II

6

SCI.ICS.L.81.1

Codification of Program Synthesis Knowledge for Concurrent Programs – Year II

Cordell Green
Principal Investigator
Computer Science Department
Systems Control, Inc.
1801 Page Mill Road
Palo Alto, CA 94304

Prepared By:
Daniel Chapiro

September 1981

FINAL TECHNICAL REPORT

Prepared for

Air Force Office of Scientific Research
Building 410
Bolling AFB, DC 20332

This research has been sponsored by the Air Force Office of Scientific Research (AFSC), United States Air Force, under contract F49620-79-C-0137. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research of the U.S. Government.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
AFOSR-TR- 81-0895 2. GOVT ACCESSION NO. AD-A110 224	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Codification of Program Synthesis Knowledge for Concurrent Programs - Year II	5. TYPE OF REPORT & PERIOD COVERED Final Report July 1, 1979-Sept. 30, 1981
	6. PERFORMING ORG. REPORT NUMBER SCI-ICS.L.81-1
7. AUTHOR(s) Daniel Chapiro	8. CONTRACT OR GRANT NUMBER(s) F49620-79-C-0137
9. PERFORMING ORGANIZATION NAME AND ADDRESS Systems Control, Inc. 1801 Page Mill Road Palo Alto, CA 94304	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Director of Mathematical and Information Sciences Air Force Office of Scientific Research Bldg. 410, Bolling AFB, DC 20332	12. REPORT DATE September 1981 13. NUMBER OF PAGES 13
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Automatic Programming, Knowledge-Based Programming, Concurrent Programming, Program Synthesis, Software Engineering, Artificial Intelligence, Highlevel Microprogramming, Microprogram Optimization, Parallelism Detection, Computer Architecure, Levels of Abstraction.

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report is the final report of our research project on the codification of concurrent programming knowledge. This project covers the period from July 1, 1979 to September 31, 1981.

The general goal of research in this area is to codify programming knowledge and to create programming systems that employ this knowledge to assist in various programming activities including specification, synthesis, modification, debugging, and maintenance. This project is concerned with the codification

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

41

of programming knowledge for concurrent programming. ~~Our~~ The aim is to produce knowledge-based design tools to help with problems in this area.

This paper primarily raises some questions that must be addressed in a study of a more focused area, namely that of generation of concurrent microcode.

We first introduce a basic parallelism operator. The intent is to refine parallel programs specified using this operator into microcode. We discuss briefly how the hardware architecture affects the level of parallelism exploited in the microcode. Then we discuss issues in the automatic generation of compact yet fast microcode. Some advantages of microcode programming by refinement of high-level specifications are brought up, namely exploiting high-level parallelism, and assurance of correctness of the resulting code. The refinement paradigm requires intermediate level constructs and search for efficient implementations, which are discussed. An example is devised to see if macroparallelism in the high-level specification is carried over in the microcode.

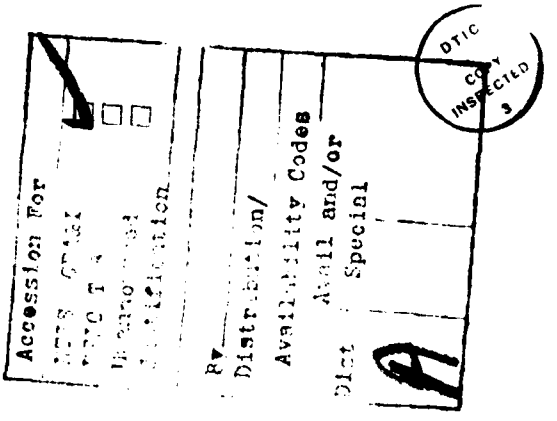
Abstract

This report is the final report of our research project on the codification of concurrent programming knowledge. This project covers the period from July 1, 1978 to September 31, 1981.

The general goal of research in this area is to codify programming knowledge and to create programming systems that employ this knowledge to assist in various programming activities including specification, synthesis, modification, debugging, and maintenance. This project is concerned with the codification of programming knowledge for concurrent programming. Our aim is to produce knowledge-based design tools to help with problems in this area.

This paper primarily raises some questions that must be addressed in a study of a more focused area, namely that of generation of concurrent microcode.

We first introduce a basic parallelism operator. The intent is to refine parallel programs specified using this operator into microcode. We discuss briefly how the hardware architecture affects the level of parallelism exploited in the microcode. Then we discuss issues in the automatic generation of compact yet fast microcode. Some advantages of microcode programming by refinement of high-level specifications are brought up, namely exploiting high-level parallelism, and assurance of correctness of the resulting code. The refinement paradigm requires intermediate level constructs and search for efficient implementations, which are discussed. An example is devised to see if macroparallelism in the high-level specification is carried over in the microcode.



§1 Introduction

This research is concerned with the building of hardware, firmware, and software tools for the design of efficient, reliable, and maintainable computing systems. These tools can use knowledge about the programming process to assist the system builder with many tasks, including design, specification, coding, debugging, modification, documentation, maintenance, and reliability assurance, all in a cost-efficient manner [Parnas] [Phillips]. Our current research deals mainly with knowledge-based concurrent programming and the process of transforming high-level specifications into microcode.

In [Green], we presented some very-high-level constructs with which we could straightforwardly specify problems that had different degrees of concurrency. We also presented a calculus which allowed us to transform these specifications into different, but equivalent, programs. The calculus consisted of a set of transformation rules which enabled us to go from the specification to different lower-level target expressions.

In this report, we consider the use of the successive refinement paradigm to transform very-high-level program specifications into the microcode target level, thus integrating the software and firmware development process. This integration requires the extension of the body of transformation rules from the domain of very-high-level expressions down to that of microcode. The use of the knowledge-based transformational approach should diminish the conflict between efficiency and ease of specification. It may also reduce some of the exponential search problems posed by microcode optimization [Dasgupta79], through the use of information carried down during transformation from higher abstraction levels. For example, the high-level specification indicates which pieces of microcode are completely independent and can be run in parallel, thus avoiding search that only re-discovers possible parallelism. Also, a knowledge-based system using stepwise refinement of high-level specifications allows efficiency estimation of alternative implementations, avoiding blind search through the space of the possible clusterings of micro-operations.

1.1 Summary of Previous Work

Most of our work has been covered in our previous report [Green]. Below we provide a brief summary of that report.

The report covered the extension of our work in sequential programming into the area of concurrent programming. First, we extended our language for express-

ing program synthesis rules to allow the expression of concurrent programming knowledge. Given that extension, we were able to express elementary transformation rules that map a high-level program (which is non-committal with respect to sequentiality or parallelism) into various concurrent versions. We gave an example in which we synthesized several versions of a simple retrieval program, with the versions having different degrees of concurrency.

An example we studied in detail is a concurrent odd-even transpose sort, which is a type of sorting network. Transpose sort did not appear to fit directly into our stepwise refinement paradigm for program construction. We proved the correctness of the algorithm, and presented a derivation which suggests the need for extending the stepwise refinement paradigm.

We briefly examined more complex algorithms including concurrent shortest path and prime-finding algorithms. These derivations appear to be as tractable as their sequential versions of comparable complexity.

The report included background material on our general approach to program synthesis. A tutorial explained how knowledge can be codified as a collection of rules which may be used to transform a specification into a program. Also discussed was the subject of which allowable reorderings of computations would not violate constraints imposed by the hardware or by the high-level specifications.

§2 Parallel Constructs

In [Green], we proposed some basic constructs which allowed us to specify highly parallel problems [Fuller]. The basic parallelism operator is //, which is applied to a function and a collection of arguments. // applies the function to each of the elements of the collection, but in any order, or in parallel.

The result returned by the // statement consists of a collection of the results of each application. If the given collection of arguments is a set, the results will be encapsulated in a set. If they are given as a list, the order of the results will be preserved as in the given list of arguments. We delimit collections with angle brackets when we talk about abstract collections, but we use curly brackets or parentheses for sets or lists respectively.

// can be terminated in three ways: (1) return a value as in standard LISP; (2) return EMPTY (distinct from NIL); and (3) abort all other parallel computations generated by this // and return a single value. For example,

$$(// \; op \; \langle arg_1 \; arg_2 \cdots arg_n \rangle)$$

would cause the parallel application of *op* to each arg_i, and the result would be

$$\langle (op\ arg_1)(op\ arg_2)\cdots \rangle.$$

If any of the applications of *op* returns EMPTY, that value will simply not appear in the result list. This enables us to represent filters. We also use a feature which allows us to stop every sibling process when one of these processes says (*Alldone* (*Value*)). This feature is particularly useful for parallel searches, because it allows us to stop all the parallel processes which are seeking some goal as soon as one of them attains it.

The way to start multiple processes is:

$$(//\ Identity\ \langle proc_1 \cdots proc_n \rangle).$$

Processes are independent computations, and no extra operator is applied once the arguments are evaluated. That is, once the processes have run (each one deciding by itself if it returns any value or not), the collection of their results becomes the result of the // statement. To simplify the notation, when the operator is just *Identity*, it need not be written. Hence (// ⟨*collection of processes*⟩) will start a group of parallel processes and terminate when one of them flags *Alldone*, or when all the processes are done normally.

On the other hand, we may want to apply the same function as a filter to all the elements of a set. This can be specified as:

$$(//\ filter\ \langle collection\ of\ elements \rangle)$$

This provides a multi fork-join facility. There is no need for more concurrency control for the so called "internal concurrency problems." Note that the join is well defined because all the arguments have to be evaluated so that the operator may be applied.

§3 Implementation with Microcode

Our calculus must be extended with new rules to transform programs from // constructs down to the microcode level. STRUM [Patterson] and S^* [Dasgupta78] provide useful ideas high-level microcode constructs.

The hardware architecture could facilitate the development of refinement derivations by providing features which resemble the higher level constructs. For example, [Arden] presents the MP/C approach, a concurrent computing environment having the shared memory aspects of tightly-coupled multiprocessors and also the characteristics usually associated with loosely-coupled message-oriented

systems. A large address space is dynamically partitioned in a hierarchical way through the use of dedicated switches which control the common bus through which the processors can access segments of a linear memory. These switches are also connected with a command bus which enables them to implement powerful operations like the ones needed for the // construct in a direct way. Hence, better use of the processors could be made through the non-commital very-high-level specification of the programs.

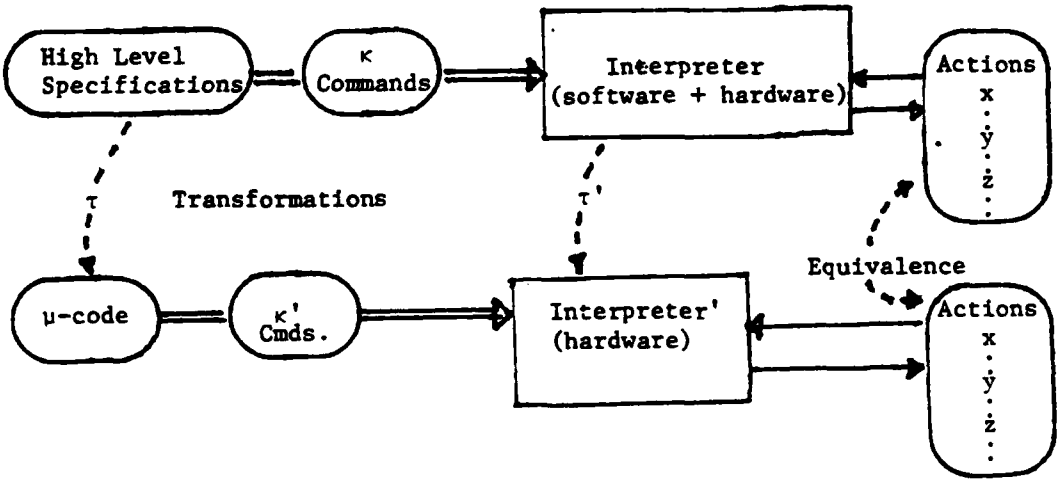


Fig.1: Transformation and interpretation

3.1 Space and Time Efficiency Considerations

Since microcode affects every operation, it cannot be inefficient. Microcode can be viewed as being the innermost loop of every computation, so that if the microcode is inefficient, every computation is inefficient. Therefore, the microcode synthesized should approach hand-coded efficiency.

A compaction problem arises when the high-level microcode is transformed into elementary control words. The micro-compilers for general purpose machines first generate sequences of micro-operations which have some constraints (introduced as a partial ordering in time and limited resources on which to run these micro-operations). Then, these operations are bundled in control words

in the most compact way which satisfies the aforementioned constraints. This compaction problem has been solved only with minor success in the past: when the microcode is mostly vertical, microcompilation as well as verification is not difficult. However, when the control words are horizontal, and there is an inherently high degree of microparallelism, the problem grows rapidly with the size of the input.

The advantage of handcoded microcode is that it can be very fast, but it is very difficult to ensure its correctness. The advantage of automatically generated microcode is that it can be proven correct with respect to higher-level specifications. Its disadvantage is that until now no one has generated automatically acceptably efficient microcode for horizontal microarchitectures.

One possibly bad approach is to generate microcode automatically and then try to optimize it. There are two key aspects to be optimized: time and space. Clearly, these are interdependent. Reducing space may be achieved by reducing either the size or the number of control words. Increasing speed requires a high degree of microparallelism and short decoding delays for the control words. These last two considerations are in conflict with the reduction of word size because a tighter encoding organization allows fewer possibilities for microparallelism. Also, the requirements of more complex decoding introduce an additional delay in the execution of each microprogram.

There is a disadvantage in using a maximal encoding schema, outside of speed considerations. Namely, adding a new instruction requires a new state and new encoding and decoding logic, all of which implies hardware change. In the maximal parallelism alternative, as many micro-operations can be run simultaneously as the hardware allows (that is, as many additions can be run at once as there are adders and appropriate data paths), and there is no need for any decoding at all, since each bit of the control word directly controls the hardware involved in a microoperation. Because of this, it is frequently called *direct control*.

An intermediate approach can avoid the lack of flexibility and parallelism of the first alternative and the excessive word length of the second. In this *minimally* encoded approach, a group of microoperations which are never executed together (that is, they are mutually exclusive during a microcycle) are encoded together. Hence, there is a partition of the totality of microoperations into k sets of m_i microoperations. This partitioning allows for an encoding in a total control word length $|w| = \sum_{i=1}^{k} \lceil \log_2 m_i \rceil$. This intermediate approach is the one in which the optimization difficulties appear. The problem consists of minimising the total number of bits $|w|$ satisfying a given set of disjoint operations. Although this requires a separate decoder for each field, each decoder is very simple since the fields are small.

The problem of minimizing the length of the microcode is closer to the traditional code optimization problems found in compilers, except for some timing restrictions and hardware dependencies that are intrinsic to microcode optimization. This similarity comes from the fact that in both cases there is a set of limited resources to assign to a computation, and a space of admissible solutions, with time/space tradeoffs. Of course, there are differences which make the techniques of software optimization not directly applicable to this kind of optimization.

Since microcode generated from higher-level specifications must be optimized, a transformational approach that modifies a specification along a refinement path can be very advantageous. By its mere existence, this path indicates the validity and satisfaction of all the constraints.

We should note that there are two efficiency considerations: the efficiency of the search in the space of implementations, and the efficiency of the resulting code. These are directly related because the higher the search efficiency, the more alternatives can be explored for the target code.

3.2 Correctness

The second reason to prefer the transformational approach is that of ensuring the correctness of the synthesized code. In [Patterson], it is reported how several errors were uncovered through the systematic verification of microcode generated from a higher-level specification language. It should be noted that, in our system, verification rules can again use the same rule-manipulation machinery, but most of them do not even need to appear in the system in any explicit way, since the transformation rules already fulfill that purpose by transforming correct specifications into correct microcode. Of course, there are still two requirements to ensure the correctness of the resulting microcode. The first is to show that the transformation rules preserve correctness. That is, the soundness and consistency of the synthesis system must be proved, but that need be done only once for all the programs to be generated. The second problem is to improve the chances of correctness of the specifications. We can significantly modify the chances for success by providing notations which will facilitate the specification of problems in a way consistent with our ideas of what we want to specify. It should be noted that this key issue to the true total correctness of programs is frequently neglected in traditional program correctness proving. We think that the clarity and simplicity which is achieved through very-high-level languages using such constructs as the ones we use is one step in the right direction for this problem.

For completeness, it should be noted that the equivalence between the original specifications and the target code does not yet include the termination properties of the programs. That is, although the specification of a program could be correct, and have a terminating solution, once some sequences of transformations are applied, the resulting code could no longer be guaranteed to terminate. This is a limitation of the calculus we presented, but, as was suggested in [Green], that issue could be taken care of by a heuristic technique which would handle the termination problem. While pruning the less efficient branches, the efficiency expert might also be able to eliminate most non-terminating ones.

§4 The Space of Refinements

One reason knowledge-based synthesis is appropriate is that a key aspect of it involves the generation of alternative implementations. Since the generation/use ratio is even lower for firmware than for systems software, it is viable to spend more time and resources in the generation and search of alternative implementations. This should not give false hopes: exponentially growing problems, even those that appear small, may be for all practical purposes unsolvable. Microcode optimization has been shown to be NP hard, which is one more reason to prefer a knowledge-based heuristic approach.

In a rule-based system, instead of having to create a complex algorithm which will generate valid microcode, the constraints are incorporated as rules in a homogeneous system of specification transformations. In this way, optimization can occur in small intermediate steps. Every branch need not be improved, since many of them will be discarded anyway. This avoids leaving all the optimization for the end, when all the higher-level information will have been lost. Most of the work done in micro-compilers up to now had either used specific algorithms or blind searches, but none used a knowledge-based search for the transformation.

There are two clearly distinct domains in which parallelism takes a central role in microprograms. One could be called "micro-parallelism", which refers to the parallelism which appears in the control of the lowest level of the abstract processor. This means that several of the sub-parts of a machine instruction are done in a concurrent fashion by the micro-architecture. This is a very low-level granularity kind of concurrency, and has little similarity with software concurrency; it is similar to the parallelism of the components of an abstract electronic device.

Some of the key issues in concurrent systems involve the development of a well-defined hierarchical structure in which lower levels implement the abstractions

used at higher levels. In this structure, the firmware can provide very efficient implementations for approaches whose practicality is marginal if implemented in software, but which become theoretically and practically sound if the firmware provides an adequate support. This brings us to several important questions which relate to architecture and concurrent software development. Mainly, what should be put in hardware, what in firmware, and what in software? This presents still another question which is how to describe and formalize what to implement at the different levels?

A system can be built by describing a sequence of refinements, in which each level supports the abstractions defined one level above by providing an adequate set of primitives which are used to define these abstractions. It is necessary to introduce islands in the language in which this refinement chain is expressed. The need for these islands arises from the significantly different kind of problems solved at each level.

Analogously, different constructs are used at each level to describe algorithms throughout their refinement from software to hardware. At each level of refinement, the formalism should be clear in its meaning. That is, it is not enough to have just a clear high-level description which is transformed into unintelligible code at lower levels. That there are different constructs in which the domain problems have to be posed at each level does not preclude a uniform approach for the refinement methodology.

Since there are various domains of objects which are populated by different kind of entities (e.g. registers, data paths, gates, control words, activation records, sets, etc.), descriptions are needed of the constraints on the interactions between these components to generate the specified behavior of the system, and also the constraints which define the valid (or even possible) modes of behavior. For example, at the software level, the set of entry points to a class gives the only possible ways to interact with it. At the firmware level, there are constraints on the number of possible parallel actions that can be initiated in a cycle because of the chosen grouping and decoding scheme. At the hardware level, these entities might be registers, data paths, etc.

This description system needs to be rich enough to describe any part for which the synthesis system is to generate options for implementation. On the other hand, it must be simple enough so that these descriptions can be manipulated easily (either manually or automatically). Earlier efforts which developed high-level microcode languages with microcompilation and/or verification in mind can suit our synthesis needs.

The verification of programs a *posteriori* is a much more difficult task than creat-

ing them correct (through correctness preserving transformations), so particular advantage can be taken of the decoupling between many of the functions which result from software migration into hardware. Most of the programs which have been *automatically* synthesized to date are relatively short, but this is precisely the case with microprograms: they tend to be short and have little interaction with other modules. There is another domain where microprogramming has a bearing on concurrency, and which corresponds to large granularity parallelism: communicating microprocesses, running on a net of microprocessors. Here, the degree of variability of the micro-substrates is even wider than for micro-architectures, and for the most part there is no developed formalism to deal with this at the substrate description level. On the other hand, there are valid analogies with the tools developed for dealing with concurrency at the software level, and which are being used with increasing success in the operating systems area.

Although we should use the simplest and most abstract notation to achieve a good degree of generality, there are some limitations which are again intrinsic to microprogramming. Namely, it makes sense to talk about a software language which runs on an abstract machine which implements the abstractions the language uses (via a compiler or an interpreter running on some extended machine [Dasgupta78]), but this cannot be used at the microprogramming level, since levels of abstraction cannot be interposed unless they are implemented in the hardware micro-architecture. Hence, we should extend the work which develops tools to describe the full scope of variability of this domain, by writing high-level microprograms which exploit the full possibilities of concurrency given by the micro-architecture.

§5 Mapping Hl Constructs into Microcode: an Example

We experimented with transforming manually a high-level expression into microcode. We assumed a simple target architecture which was devoid of high-level parallelism, but which allowed several microoperations to take place in the same cycle. There were several registers, which allowed overlapped fetch and store, but only one ALU which limited the available parallelism for arithmetic operations. The intention was to find if a system could take advantage of the microparallelism when the specification of the problem allowed high level concurrency. The specification was

$$\phi(struct) = \{min(x) \mid x \in struct\}$$

which means to find the set composed of the minimums of each of the subsets which are in *struct*. For example: $\phi(\{\{1\ 3\ 6\}\{87\ 12\}\}) = \{1\ 12\}$. This is non-

committal as to a parallel or serial implementation. Noticing that the operation *min* in each of the sets does not affect the value of the elements of the set, and that to calculate *min* of a set only the values of the elements of the set are needed, implies that the *min* of for all the subsets can be computed in parallel. Hence, using our // notation, the specification may be transformed into:

(// *min struct*)

which could be computed in INTERLISP with:

(*for s in struct collect* (*min s*)).

Of course, in neither of the last two transformations is there any indication of the order in which these *min* operations are to be performed. Next, we transformed this into a serial algorithm, then into a pidgin high-level microcode language, with the expectation of being able to take advantage of all the microparallelism features of the assumed architecture. At this point, the higher-level parallelism could not be extracted (or detected) from looking at the serial algorithm microcode. This seems to support our belief that the parallelism of an algorithm can be better exploited at a high level. That is, there is no clear way in which we could see the macroparallelism through microparallelism. It should be noted that in spite of this, many special-purpose machines may have advantages at the microparallelism level, in particular for very regular numerical algorithms.

§6 Conclusions

We have brought out some issues concerning the use of the paradigm of stepwise refinement to transform concurrent program specifications using very-high-level constructs into microcode. We considered optimization in the light of this paradigm, and how it could give us an edge over a strictly algorithmic approach. Work is needed to select appropriate intermediate-level constructs, and to develop further the base of refinement rules that code high-level parallelism.

Acknowledgements: Cordell Green, Beverly Kedzierski, and Tom Pressburger helped in editing this report.

§7 References

[Arden] Arden, B. and Ginosar, R. "MP/C: A Multiprocessor/Computer Architecture," in Conference Proceedings of the 8-th Annual Symposium of Computer Architecture. Dept. of Electrical Engineering and Computer Science, Princeton University, New Jersey, 1981.

[Dasgupta78] Dasgupta, S. "Towards a microprogramming language schema," in Proceedings of the 11-th Annual Workshop on Microprogramming (ACM), Nov 1978, pp. 144-153.

[Dasgupta79] Dasgupta, S. "The Organization of Microprograms Stores," *Computing Surveys 11*, 1 (March 79), pp. 40-65.

[DeWitt] DeWitt, D.J. "Extensibility: A New Approach For Designing Machine Independent Programming Languages," in Proceedings of the 9-th Annual Workshop on Microprogramming (ACM), Sept. 1976, pp. 33-41.

[Fuller] Fuller S., Lesser V., Bell G. and Kaman C. "The Effects of Emerging Technology and Emulation Requirements on Microprogramming," *IEEE Transactions on Computers C-25*, 10 (Oct. 1976), pp. 1000-1009.

[Green] Green, C., Chapiro, D., and Pressburger, T. "Codification of Concurrent Programming Knowledge", Systems Control Inc., Palo Alto, California, SCI Technical Report SCI-ICS L.81.1., November 1980.

[Parnas] Parnas, D. L. and Siewiorek, D. P. "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," *Communications of the ACM 18*, 7 (July 1975), pp. 401-408.

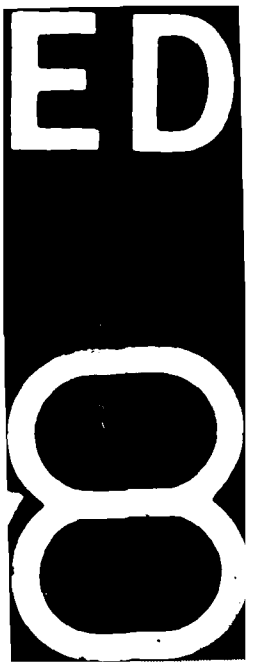
[Patterson] Patterson, D. A. "STRUM: Structured Programming Systems for Correct Firmware", *IEEE Transactions on Computers C-25*, 10 (Oct. 1976), pp. 974-985.

[Phillips] Phillips, J. and Green, C. "Towards Self Described Programming Environments", Systems Control Inc., Palo Alto, California Technical Report SCI.ICS.U.80.1, June 1980.

[Robertson] Robertson, E. "Microcode Bit Optimization is NP-Hard," *Sigmicro Newsletter 8*, 2 (June 1977), pp. 40-43.

[Tsuchiya] Tsuchiya, M. and Gonzalez, M. "Toward Optimization of Horizontal Microprograms," *IEEE Transactions on Computers C-25*, 10 (Oct.,

1976), pp.992-999.


ED
8